LM Maze router algorithm implementation code

MOHIT SHARMA
110031631

Header files:

Node.h

```
// Taken from https://github.com/abangfarhan/graph-sfml/blob/master/include/Node.h

#ifndef NODE_H
#define NODE_H
class Node {
      public:
            Node();
            Node(float x, float y, std::string name = "");
            Node(Node* node);
            float x();
            float y();
            std::string name();
            void setX(float x);
            void setY(float y);
            void setName(std::string name);
            void addNeighbor(Node* node);
            std::vector<Node*> neighbors();
            float distance(Node* neighbor);
      private:
            float _x;
            float _y;
            std::string _name;
            std::vector<Node*> _neighbors;
};
#endif
```

NodeLM.h

```
class NodeLM: public Node {

     public:
        NodeLM(float x, float y, std::string name = ""): Node(x, y, name) {}
        NodeLM() {}
        void setLabel(int label);
        int getLabel();
        bool is_obstruction();
        void setnodeasobstruction();
```

```
        std::vector<NodeLM*> neighbors();
        void addNeighbor(NodeLM* node);
        void update_line_exists_with_neighbor_node(NodeLM*,int);
        void update_has_source_node(int);
        int is_source_node();
        int is_target_node();
        void update_has_target_node(int);
        int is_to_be_shown_connected_to(NodeLM*);
        std::string get_direction(int,NodeLM*,std::string,std::string);
        int get_num_bends(int,NodeLM*,std::string,std::string);
        std::vector<std::pair<std::pair<std::string,std::string>,NodeLM*>>
get_path(int);
        void add_prev_node_data(int, std::string, std::string, NodeLM*,
std::string, int, int, int);
        void delete_prev_node_data();
        int is_already_on_a_nets_path(int);
        int is_already_on_a_selected_path();
        void add_on_selected_path();
        void delete_from_selected_path();
        std::pair<std::string,NodeLM*> get_path_with_min_bends(int,
std::string);
        std::pair<int,int> get_coordinates(int);
        int get_node_num(int, int);
     private:
        int _label = 0;
        bool _is_obstruction = false;
        std::vector<std::pair<NodeLM*,int>> _neighbors;
        int _is_on_selected_path;
        int _has_source_node = 0;
        int _has_target_node = 0;
std::vector<std::tuple<int,std::pair<std::string,std::string>,NodeLM*,std::strin
g,int,int>> _path_data;
};
```

---

read_data_helper.h

```
#include <string>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <regex>

NodeLM* read_maze_file(std::string maze_file, int * gridsize,
std::vector<std::pair<std::pair<int,int>,std::pair<int,int>>> *nets, int
grid_node_size, int debug) {
    NodeLM* nodeList;
    std::string line = "";
    int x_coordinate, y_coordinate, source_x_coordinate, source_y_coordinate,
dest_x_coordinate, dest_y_coordinate;

    std::ifstream file(maze_file.c_str());
    while (line.find("grid size") == std::string::npos) {
      getline(file,line);
    }
    std::regex pattern ("(\\d*).*");
    std::smatch match;
```

```
    // Read grid size
    bool pattern_matched = std::regex_search (line, match, pattern);
    if ( pattern_matched ) {
        *gridsize = atoi(match[1].str().c_str());
        nodeList = new NodeLM[(*gridsize) * (*gridsize)];
    }
    if ( debug ) { std::cout << (*gridsize) * (*gridsize) << std::endl; }
    while ( line.find("obstruction") == std::string::npos) {
      getline(file,line);
    }

    // Read obstruction nodes
    pattern = ("obstruction (\\d*) (\\d*)");
    while ( line.find("obstruction") != std::string::npos) {
      pattern_matched = std::regex_search (line, match, pattern);
      if ( pattern_matched ) {
         x_coordinate = atoi(match[1].str().c_str());
         y_coordinate = atoi(match[2].str().c_str());
      }
      nodeList[x_coordinate * (*gridsize) +
y_coordinate].setnodeasobstruction();
      if ( debug ) { std::cout << x_coordinate << ":" << y_coordinate << "\n";}
      getline(file,line);
    }

    while ( line.find("net") == std::string::npos) {
      getline(file,line);
    }

    // Read source and destination nodes for all nets
    pattern = ("net *(\\d*) *(\\d*) *(\\d*) *(\\d*)");
    while ( line.find("net") != std::string::npos) {
      pattern_matched = std::regex_search (line, match, pattern);
      if ( pattern_matched ) {
         source_x_coordinate = atoi(match[1].str().c_str());
         source_y_coordinate = atoi(match[2].str().c_str());
         dest_x_coordinate = atoi(match[3].str().c_str());
         dest_y_coordinate = atoi(match[4].str().c_str());
      }

(*nets).push_back(std::make_pair(std::make_pair(source_x_coordinate,source_y_coo
rdinate),std::make_pair(dest_x_coordinate,dest_y_coordinate)));
      if ( debug ) { std::cout << source_x_coordinate << ":" <<
source_y_coordinate << "::" << dest_x_coordinate << ":" << dest_y_coordinate <<
"\n";}
      getline(file,line);
    }


    file.close();
    return nodeList;
}
```

LM_graphHelper.h

```
#include <math.h>
#include <time.h>

#define PI 3.14159265
```

```
sf::CircleShape Triangle(float x1, float y1, float x2, float y2, float radius) {
     sf::CircleShape triangle(radius, 3);
     float angle;

     if ( x1  == x2 )
        if ( y1 < y2 ) {
           angle = 180;
           triangle.setPosition(x2 + 3*radius/4,y2 + 1*radius/4);
        } else {
           angle = 0;
           triangle.setPosition(x2 - 3*radius/4,y2 - 3*radius/4);
        }
     else
        if ( x1 < x2 ) {
           angle = 90;
           triangle.setPosition(x2 - 1*radius/4,y2 - 3*radius/4);
        } else {
           angle = 270;
           triangle.setPosition(x2 - 1*radius/4,y2 + 3*radius/4);
        }
     triangle.setRotation(angle);
     return triangle;
}

sf::RectangleShape Line(float x1, float y1, float x2, float y2, float thickness
= 1) {
     sf::RectangleShape line;
     float len, angle;

     line.setPosition(x1, y1);
     len = sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
     angle = atan((y2 - y1) / (x2 -x1)) * 180 / PI;
     // line pointing down-left and top-left must be incremented by 180 deg
     if ( x2 - x1 < 0 ) angle += 180;

     line.setSize(sf::Vector2f(len, thickness));
     line.setFillColor(sf::Color(100, 100, 100));
     line.setRotation(angle);
     return line;
}

void drawtext(float x, float y, int charsize, sf::Color txt_color, sf::Font
*font, std::string txt_string, sf::RenderWindow *window) {
     sf::Text txt;
     txt.setPosition(x, y);
     txt.setCharacterSize(charsize);
     txt.setColor(txt_color);
     txt.setFont((*font));
     txt.setString(txt_string);
     (*window).draw(txt);
}

void drawGraph(int n_nodes, NodeLM * nodeList, int grid_node_size, int gridsize,
NodeLM s, NodeLM t) {
     sf::RectangleShape nodesqrs[n_nodes];
     int width = grid_node_size * gridsize;
     int height = grid_node_size * gridsize;
     for (int i = 0; i < n_nodes; ++i) {
        nodesqrs[i].setPosition(nodeList[i].x() -
grid_node_size/2,nodeList[i].y() - grid_node_size/2);
        nodesqrs[i].setOutlineColor(sf::Color(0,0,0));
        nodesqrs[i].setSize(sf::Vector2f(grid_node_size, grid_node_size));
        nodesqrs[i].setOutlineThickness(1);
        if ( nodeList[i].is_obstruction() ) {
```

```
            nodesqrs[i].setFillColor(sf::Color(0, 0, 0));
         } else {
            nodesqrs[i].setFillColor(sf::Color(255,255,255));
         }
      }

      int radius = 3;

      sf::CircleShape startNode;
      startNode.setPosition(s.x() - radius, s.y() - radius);
      startNode.setRadius(radius);
      startNode.setFillColor(sf::Color::Blue);

      sf::CircleShape endNode;
      endNode.setPosition(t.x() - radius, t.y() - radius);
      endNode.setRadius(radius);
      endNode.setFillColor(sf::Color::Red);

      sf::ContextSettings settings;
      settings.antialiasingLevel = 8;
      sf::RenderWindow window(sf::VideoMode(width, height), "Graph",
sf::Style::Default, settings);
      while (window.isOpen()) {
       sf::Event event;
       while (window.pollEvent(event)) {
         if ( event.type == sf::Event::Closed) {
            window.close();
         }
       }
       window.clear(sf::Color::White);
       for (int i = 0; i < n_nodes; ++i) {
           window.draw(nodesqrs[i]);
           window.draw(startNode);
           window.draw(endNode);
       }
       window.display();
     }
}

void fillGraphGrid (NodeLM* nodeList, int gridsize, int grid_node_size) {
      int num_nodes, next_element, previous_element, next_row, previous_row;

      num_nodes = gridsize * gridsize;
      for (int i=0; i < num_nodes; i++) {
          nodeList[i].setX(((i / gridsize) * grid_node_size) + grid_node_size +
grid_node_size/2);
          nodeList[i].setY(((i % gridsize) * grid_node_size) + grid_node_size +
grid_node_size/2);
          previous_row = i - gridsize;
          next_row = i + gridsize;
          next_element = i + 1;
          previous_element = i - 1;
          if ( previous_row >= 0 )
             nodeList[i].addNeighbor(&nodeList[previous_row]);
          if ( next_row < num_nodes )
             nodeList[i].addNeighbor(&nodeList[next_row]);
          if ( previous_element > 0 && ((previous_element / gridsize) == (i /
gridsize)) )
             nodeList[i].addNeighbor(&nodeList[i-1]);
          if ( next_element < num_nodes && ((next_element / gridsize) == (i /
gridsize)) )
             nodeList[i].addNeighbor(&nodeList[i+1]);
      }
```

```
}


void initialize_nodes_to_draw(sf::RectangleShape *nodesqrs, NodeLM *nodeList,
sf::Text *text, sf::Font *font, int n_nodes, int grid_node_size) {
  for (int i = 0; i < n_nodes; ++i) {
     nodesqrs[i].setPosition(nodeList[i].x() - grid_node_size/2,nodeList[i].y()
- grid_node_size/2);
     nodesqrs[i].setOutlineColor(sf::Color(0,0,0));
     nodesqrs[i].setSize(sf::Vector2f(grid_node_size, grid_node_size));
     nodesqrs[i].setOutlineThickness(1);
     if ( nodeList[i].is_obstruction() ) {
        nodesqrs[i].setFillColor(sf::Color(0, 0, 0));
     } else {
        nodesqrs[i].setFillColor(sf::Color(255,255,255));
     }
     text[i].setPosition(nodeList[i].x() - grid_node_size/4, nodeList[i].y() -
grid_node_size/4);
     text[i].setCharacterSize(20);
     text[i].setColor(sf::Color::Black);
     text[i].setFont((*font));
  }
}

void draw_boundary_nodes_and_starting_grid (sf::RenderWindow *window,
sf::RectangleShape *nodesqrs, sf::Font *font, int radius, int gridsize, int
grid_node_size, int gridwidth, int gridheight, int n_nodes) {
  int i;
  for (int index = 0; index <= n_nodes + (2*gridsize - 1); ++index) {
    i = index - (gridsize + 1) - (index/(gridsize +1));
    if ( index <= gridsize  || index % (gridsize + 1) == 0) {
       sf::RectangleShape boundary_node;
       int x_coordinate = (((index%(gridsize+1)) * grid_node_size));
       int y_coordinate = (((index/(gridsize+1)) * grid_node_size));
       boundary_node.setPosition(x_coordinate,y_coordinate);
       boundary_node.setOutlineColor(sf::Color(0,0,0));
       boundary_node.setOutlineThickness(1);
       boundary_node.setSize(sf::Vector2f(grid_node_size, grid_node_size));
       boundary_node.setFillColor(sf::Color(255, 255, 0));
       (*window).draw(boundary_node);
       if ( !index ) {
          drawtext(x_coordinate + (2.5 *grid_node_size)/4, y_coordinate +
grid_node_size/10, 20, sf::Color::Black, font, "x", window);
          drawtext(x_coordinate + grid_node_size/8, y_coordinate +
(3*grid_node_size)/8, 20, sf::Color::Black, font, "y", window);
          sf::RectangleShape line = Line(0,0,grid_node_size,grid_node_size,2);
          (*window).draw(line);
       } else if ( index <= gridsize ) {
          drawtext(x_coordinate + grid_node_size/4, y_coordinate +
grid_node_size/4, 20, sf::Color::Black, font, std::to_string(index - 1),
window);
       } else {
          drawtext(x_coordinate + grid_node_size/4, y_coordinate +
grid_node_size/4, 20, sf::Color::Black, font, std::to_string((index/
(gridsize+1)) - 1), window);
       }
    } else {
       if ( i >= 0 )
          (*window).draw(nodesqrs[i]);
    }
  }
  sf::RectangleShape line = Line(gridwidth + 3.5,0,gridwidth +
3.5,gridheight,7);
  line.setFillColor(sf::Color(0,0,0));
```

```
  (*window).draw(line);
  sf::CircleShape greenCircle_legend;
  greenCircle_legend.setPosition(gridwidth + grid_node_size,
(gridsize*grid_node_size / 5));
  greenCircle_legend.setRadius(radius);
  greenCircle_legend.setFillColor(sf::Color::Green);
  (*window).draw(greenCircle_legend);
  drawtext(gridwidth + 2*grid_node_size, (gridsize*grid_node_size / 5), 20,
sf::Color::Black, font, "Current source & target nodes being routed", window);
  sf::CircleShape redCircle_legend;
  redCircle_legend.setPosition(gridwidth + grid_node_size, (2 *
gridsize*grid_node_size / 5));
  redCircle_legend.setRadius(radius);
  redCircle_legend.setFillColor(sf::Color::Red);
  (*window).draw(redCircle_legend);
  drawtext(gridwidth + 2*grid_node_size, (2 * gridsize*grid_node_size / 5), 20,
sf::Color::Black, font, "Target node", window);
  sf::CircleShape blueCircle_legend;
  blueCircle_legend.setPosition(gridwidth + grid_node_size, (3 *
gridsize*grid_node_size / 5));
  blueCircle_legend.setRadius(radius);
  blueCircle_legend.setFillColor(sf::Color::Blue);
  (*window).draw(blueCircle_legend);
  drawtext(gridwidth + 2*grid_node_size, (3 * gridsize*grid_node_size / 5), 20,
sf::Color::Black, font, "Source node", window);
  sf::RectangleShape obstruction_legend;
  obstruction_legend.setPosition(gridwidth + grid_node_size , (4 *
gridsize*grid_node_size / 5));
  obstruction_legend.setOutlineColor(sf::Color(0,0,0));
  obstruction_legend.setOutlineThickness(1);
  obstruction_legend.setSize(sf::Vector2f(grid_node_size, grid_node_size));
  obstruction_legend.setFillColor(sf::Color(0, 0, 0));
  (*window).draw(obstruction_legend);
  drawtext(gridwidth + 3*grid_node_size, (4 * gridsize*grid_node_size / 5), 20,
sf::Color::Black, font, "Obstruction node", window);
}

void draw_previous_nets_sources_and_sinks (sf::RenderWindow *window, NodeLM
*nodeList, int radius, int n_nodes) {
  for (int i = 0; i < n_nodes; ++i) {
    if (nodeList[i].is_source_node()) {
       sf::CircleShape startNode;
       startNode.setPosition(nodeList[i].x() - radius, nodeList[i].y() -
radius);
       startNode.setRadius(radius);
       startNode.setFillColor(sf::Color::Blue);
       (*window).draw(startNode);
    }
    if (nodeList[i].is_target_node()) {
       sf::CircleShape endNode;
       endNode.setPosition(nodeList[i].x() - radius, nodeList[i].y() - radius);
       endNode.setRadius(radius);
       endNode.setFillColor(sf::Color::Red);
       (*window).draw(endNode);
    }
    for (NodeLM* neighbor : nodeList[i].neighbors()) {
        if (nodeList[i].is_to_be_shown_connected_to(neighbor)) {
           //std::cout << nodeList[i].x() << ":" << nodeList[i].y() << "::" <<
neighbor->x() << ":" << neighbor->y() << "\n";
           sf::RectangleShape line = Line(nodeList[i].x(), nodeList[i].y(),
neighbor->x(), neighbor->y(), 5);
           line.setFillColor(sf::Color(255,255,0));
           sf::CircleShape triangle = Triangle(nodeList[i].x(), nodeList[i].y(),
neighbor->x() , neighbor->y(), 7);
```

```
            triangle.setFillColor(sf::Color(0,0,0));
            (*window).draw(line);
            (*window).draw(triangle);
          }
      }
    }
}

void draw_current_source_and_sink (NodeLM *s, NodeLM *t, int radius,
sf::RenderWindow *window, sf::Color source_color, sf::Color target_color) {
  sf::CircleShape startNode;
  startNode.setPosition(s->x() - radius, s->y() - radius);
  startNode.setRadius(radius);
  startNode.setFillColor(source_color);

  sf::CircleShape endNode;
  endNode.setPosition(t->x() - radius, t->y() - radius);
  endNode.setRadius(radius);
  endNode.setFillColor(target_color);
  (*window).draw(startNode);
  (*window).draw(endNode);
  (*window).display();
}

void blink_current_source_and_sink (NodeLM* s, NodeLM* t, int radius,
sf::RenderWindow *window) {
  int ctr = 5;
  while ( ctr > 0 ) {
    draw_current_source_and_sink (s, t, radius, window, sf::Color::Black,
sf::Color::Black);
    sf::sleep(sf::milliseconds(500));
    draw_current_source_and_sink (s, t, radius, window, sf::Color::Blue,
sf::Color::Red);
    sf::sleep(sf::milliseconds(500));
    ctr--;
  }
}

void draw_traced_path(NodeLM *s, NodeLM *t, int ctr, sf::RenderWindow *window) {
  NodeLM* curr_node, *prev_node;
  std::pair<std::string,NodeLM*> min_bends_path;
  std::string dir = "none";

  curr_node = s;
  do {
    min_bends_path = curr_node->get_path_with_min_bends(ctr,dir);
    dir = min_bends_path.first;
    prev_node = min_bends_path.second;
    sf::RectangleShape line = Line(curr_node->x(), curr_node->y(), prev_node-
>x(), prev_node->y(), 5);
    line.setFillColor(sf::Color(255,255,0));
    (*window).draw(line);
    curr_node->update_line_exists_with_neighbor_node(prev_node,1);
    curr_node->add_on_selected_path();
    curr_node = prev_node;
    (*window).display();
  } while (curr_node != t);
  (*window).display();
}
```

LM_router.h

```
#include <limits>
#include <iostream>
#include <vector>
#include <set>
#include <random>
#include <string>
#include <cstddef>
#include <math.h>

#define PI 3.14159265

std::string get_dir(int node1_x, int node1_y, int node2_x, int node2_y) {
  std::string dir;
  if ( node1_x  == node2_x )
     if ( node1_y < node2_y )
        dir = "down";
     else
        dir = "up";
  else
     if ( node1_x < node2_x )
        dir = "right";
     else
        dir = "left";
  return dir;
}

void retrace(NodeLM* s, NodeLM* t, int gridsize, int grid_node_size, int ctr,
sf::RenderWindow *window) {
  std::set<NodeLM*> previous_node;
  std::set<NodeLM*> new_previous_nodes;
  //int node_num, count; *color;
  int num_bends;
  std::string path_sum_x, path_sum_y;
  std::string dir;
  previous_node.insert(t);
  do {
    for (NodeLM* e : previous_node) {
      for (NodeLM* neighbor : e->neighbors()) {
        if ( neighbor->getLabel() == (e->getLabel() - 1) ) {
           if ( e->x() == t->x() && e->y() == t->y() ) {
              e->add_prev_node_data(ctr,std::to_string((e-
>get_coordinates(grid_node_size)).first), std::to_string((e-
>get_coordinates(grid_node_size)).second),e,"none",0,0,0);
           }
           dir = get_dir(e->x(), e->y(), neighbor->x(), neighbor->y());
           for (std::pair<std::pair<std::string,std::string>,NodeLM*> path: e-
>get_path(ctr)) {
             if ( dir == e-
>get_direction(ctr,path.second,path.first.first,path.first.second) || e-
>get_direction(ctr,path.second,path.first.first,path.first.second) == "none" ) {
                num_bends = e-
>get_num_bends(ctr,path.second,path.first.first,path.first.second);
             } else {
                num_bends = e-
>get_num_bends(ctr,path.second,path.first.first,path.first.second) + 1;
             }
             path_sum_x = path.first.first + std::to_string((neighbor-
>get_coordinates(grid_node_size)).first);
             path_sum_y = path.first.second + std::to_string((neighbor-
>get_coordinates(grid_node_size)).second);
             neighbor-
```

```
>add_prev_node_data(ctr,path_sum_x,path_sum_y,e,dir,num_bends,neighbor-
>is_already_on_a_nets_path(ctr),0);
            }
            new_previous_nodes.insert(neighbor);
            sf::RectangleShape line = Line(e->x(), e->y(), neighbor->x(),
neighbor->y(), 3);
            //std::cout << neighbor->x() << ":" << neighbor->y() << "\n";
            line.setFillColor(sf::Color::Green);
            sf::CircleShape triangle = Triangle(e->x(), e->y(), neighbor->x() ,
neighbor->y(), 5);
            triangle.setFillColor(sf::Color::Green);
            (*window).draw(line);
            (*window).draw(triangle);
            (*window).display();
            sf::sleep(sf::milliseconds(25));
          }
        }
      }
      previous_node = new_previous_nodes;
      new_previous_nodes.clear();
    } while (!previous_node.empty());
}


void lm_router(std::vector<std::pair<std::pair<int,int>,std::pair<int,int>>>
*nets, NodeLM *nodeList, int n_nodes, int gridsize, int grid_node_size) {
  std::set<NodeLM*> plist;
  std::set<NodeLM*> nlist;
  NodeLM *s,*t;
  int temp = 1, path_exists = 0,  already_drawn = 0, ctr = 0;
  std::pair<std::pair<int,int>,std::pair<int,int>> tmp_value;
  int num_nets = nets->size();
  int num_tries = num_nets;
  int num_nets_successfully_routed = 0;

  srand( (unsigned)time(NULL));

  sf::RectangleShape nodesqrs[n_nodes];
  sf::Text text[n_nodes];
  sf::Font font;
  font.loadFromFile("./src/Roboto/Roboto-Regular.ttf");

  int width = grid_node_size * (gridsize+1) + 500;
  int grid_width = grid_node_size * (gridsize+1);
  int height = grid_node_size * (gridsize+1);
  int radius = 10;

  initialize_nodes_to_draw(nodesqrs, nodeList, text, &font, n_nodes,
grid_node_size);

  sf::ContextSettings settings;
  settings.antialiasingLevel = 8;
  sf::RenderWindow window;
  window.create(sf::VideoMode(width, height), "Net Routing", sf::Style::Default,
settings);

  while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
      if ( event.type == sf::Event::Closed) {
         window.close();
      }
    }
    if ( ctr < num_nets ) {
```

```
      std::vector<std::pair<std::pair<int,int>,std::pair<int,int>>>::iterator
path = (*nets).begin();
      window.clear(sf::Color::White);
      draw_boundary_nodes_and_starting_grid (&window, nodesqrs, &font, radius,
gridsize, grid_node_size, grid_width, height, n_nodes);
      draw_previous_nets_sources_and_sinks (&window, nodeList, radius, n_nodes);
      std::cout <<
"\n*****************************************************************\n";
      std::cout << "*Try number : " << num_nets - num_tries + 1<< "\n\n";
      while ( path != (*nets).end() ) {
        std::cout << "**Routing net number: " << ctr + 1 << "\n";
        for (std::pair<std::pair<int,int>,std::pair<int,int>> s_and_t_pairs:
(*nets)) {
          s = &nodeList[(s_and_t_pairs).first.first * gridsize +
(s_and_t_pairs).first.second];
          s->update_has_source_node(1);
          t = &nodeList[(s_and_t_pairs).second.first * gridsize +
(s_and_t_pairs).second.second];
          t->update_has_target_node(1);
        }

        s = &nodeList[(*path).first.first * gridsize + (*path).first.second];
        t = &nodeList[(*path).second.first * gridsize + (*path).second.second];

        plist.insert(s);
        std::cout << "\n---------------------------\n";
        std::cout << "***Source coordinates: " << (s-
>get_coordinates(grid_node_size)).first << ":" << (s-
>get_coordinates(grid_node_size)).second << "\n";
        std::cout << "***Target coordinates: " << (t-
>get_coordinates(grid_node_size)).first << ":" << (t-
>get_coordinates(grid_node_size)).second << "\n";
        std::cout << "---------------------------\n";

        draw_previous_nets_sources_and_sinks (&window, nodeList, radius,
n_nodes);
        draw_current_source_and_sink (s, t, radius, &window, sf::Color::Green,
sf::Color::Green);

        if ( !already_drawn ) {
          while (!plist.empty()) {
            for (NodeLM* e : plist) {
              for (NodeLM* neighbor : e->neighbors()) {
                if ( !neighbor->is_already_on_a_selected_path() && !neighbor-
>is_obstruction() && neighbor->getLabel() == 0 && !neighbor->is_source_node() &&
(!neighbor->is_target_node() || neighbor == t)){
                  neighbor->setLabel(temp);
                  text[neighbor-
>get_node_num(grid_node_size,gridsize)].setString(std::to_string(temp));
                  window.draw(text[neighbor-
>get_node_num(grid_node_size,gridsize)]);
                  window.display();
                  sf::sleep(sf::milliseconds(25));
                  nlist.insert(neighbor);
                  //std::cout << neighbor->x() << ":" << neighbor->y() << " ";
                  if ((neighbor->x() == t->x() ) && (neighbor->y() == t->y()))
{
                     path_exists = 1;
                     break;
                  }
                }
                if ( neighbor->is_obstruction() && neighbor->getLabel() == 0 &&
neighbor->x() == t->x() && neighbor->y() == t->y() ) {
                  neighbor->setLabel(temp);
```

```
                    path_exists = 1;
                    break;
                }
              }
              //std::cout << "\n";
            }
            if ( path_exists ) { break; }
            temp++;
            plist = nlist;
            nlist.clear();
          }
          if ( path_exists && !already_drawn ) {
             already_drawn = 1;
             std::cout << "Path exists between source and target
nodes\n_____________________________________________\n\n";
             num_nets_successfully_routed++;
             retrace(s, t, gridsize, grid_node_size, ctr, &window);
             draw_traced_path(s, t, ctr, &window);
          } else {
             if ( !path_exists ) {
                std::cout << "Path does not exist between source and target
nodes\n\n";
                std::cout <<
"\n*****************************************************************\n";
                if ( num_tries > 0 ) {
                   blink_current_source_and_sink (s, t, radius, &window);
                   num_tries--;
                   tmp_value = (*path);
                   (*nets).erase(path);
                   path = (*nets).begin();
                   (*nets).insert(path, tmp_value);
                   path = (*nets).begin();
                   ctr = 0;
                   already_drawn = 0;
                   path_exists = 0;
                   temp = 1;
                   plist.clear();
                   nlist.clear();
                   num_nets_successfully_routed = 0;
                   for ( int i = 0; i < n_nodes ; i ++ ) {
                     nodeList[i].setLabel(0);
                     nodeList[i].delete_from_selected_path();
                     nodeList[i].delete_prev_node_data();
                     nodeList[i].update_has_target_node(0);
                     nodeList[i].update_has_source_node(0);
                     for (NodeLM* neighbor : nodeList[i].neighbors()) {

nodeList[i].update_line_exists_with_neighbor_node(neighbor,0);
                     }
                   }
                   //std::cout << "\nChecking" << (*nets).size() << "\n";
                   sf::sleep(sf::milliseconds(10000));
                   window.clear(sf::Color::White);
                   draw_boundary_nodes_and_starting_grid (&window, nodesqrs,
&font, radius, gridsize, grid_node_size, grid_width, height, n_nodes);
                   draw_previous_nets_sources_and_sinks (&window, nodeList,
radius, n_nodes);
                   window.display();
                   std::cout << "*Try number : " << num_nets - num_tries + 1 <<
"\n\n";
                   continue;
                }
             }
          }
```

```
        }
        sf::sleep(sf::milliseconds(25));
        ctr++;
        already_drawn = 0;
        path_exists = 0;
        temp = 1;
        plist.clear();
        nlist.clear();
        for ( int i = 0; i < n_nodes ; i ++ ) {
          nodeList[i].setLabel(0);
        }
        path++;
        window.clear(sf::Color::White);
        draw_boundary_nodes_and_starting_grid (&window, nodesqrs, &font, radius,
gridsize, grid_node_size, grid_width, height, n_nodes);
        draw_previous_nets_sources_and_sinks (&window, nodeList, radius,
n_nodes);
        window.display();
      }
      std::cout <<
"**********************************************************************\n\n";
      std::cout <<
"**********************************************************************\n";
      std::cout << "Number of nets successfully routed = " <<
num_nets_successfully_routed << "/" << num_nets << " in try number " << num_nets
- num_tries + 1;
      std::cout <<
"\n**********************************************************************\n";
    }
  }
}
```

SRC files:

Node.cpp

```
// Taken from https://github.com/abangfarhan/graph-sfml/blob/master/src/Node.cpp

#include <vector>
#include <string>
#include <math.h>

#include "Node.h"

Node::Node() {
    setX(0);
    setY(0);
    setName("");
}

Node::Node(float x, float y, std::string name) {
    setX(x);
    setY(y);
    setName(name);
}
```

```
Node::Node(Node* node) {
    /* copy properties, except the neighbors */
    setX(node->x());
    setY(node->y());
    setName(node->name());
}

float Node::x() {
      return _x;
}

float Node::y() {
      return _y;
}

std::string Node::name() {
      return _name;
}

void Node::setX(float x) {
      _x = x;
}

void Node::setY(float y) {
      _y = y;
}

void Node::setName(std::string name) {
      _name = name;
}

void Node::addNeighbor(Node* node) {
     // Add neighbor node to this->_neighbors if not exist,
     // and add this to the neighbor node->_neighbors
     for (Node* neighbor: _neighbors)
         if (neighbor == node)
            return;
     _neighbors.push_back(node);
     node->addNeighbor(this);
}

std::vector<Node*> Node::neighbors() {
     return _neighbors;
}

float Node::distance(Node* neighbor) {
      return sqrt(pow(_x - neighbor->x(), 2) + pow(_y - neighbor->y(), 2));
}
```

NodeLM.cpp

```
#include <vector>
#include <string>
#include <utility>
#include <iostream>
```

```cpp
#include <iterator>
#include <map>
#include <climits>

#include "Node.h"
#include "NodeLM.h"

int NodeLM::getLabel() {
      return _label;
}

void NodeLM::setLabel(int label) {
     _label = label;
}

bool NodeLM::is_obstruction() {
     return _is_obstruction;
}

void NodeLM::setnodeasobstruction() {
    _is_obstruction = true;
}

std::vector<NodeLM*> NodeLM::neighbors() {
     std::vector<NodeLM*> neighbors;
     std::vector<std::pair<NodeLM*,int>>::iterator neighbor;
     for (neighbor = _neighbors.begin(); neighbor!=_neighbors.end(); neighbor++)
        neighbors.push_back((*neighbor).first);
     return neighbors;
}

void NodeLM::addNeighbor(NodeLM* node) {
     for (std::pair<NodeLM*,int> neighbor: _neighbors)
         if (neighbor.first == node) {
            neighbor.second = 0;
            return;
         }
     _neighbors.push_back(std::make_pair(node,0));
     node->addNeighbor(this);
}

void NodeLM::update_line_exists_with_neighbor_node(NodeLM* node, int value) {
     std::vector<std::pair<NodeLM*,int>>::iterator neighbor;
     for (neighbor = _neighbors.begin(); neighbor!=_neighbors.end(); neighbor++)
         if ((*neighbor).first == node)
            (*neighbor).second = value;
}

int NodeLM::is_to_be_shown_connected_to(NodeLM* node) {
    for (std::pair<NodeLM*,int> neighbor: _neighbors)
        if (neighbor.first == node)
           return neighbor.second;
    return 0;
}

void NodeLM::update_has_source_node(int value) {
     _has_source_node = value;
}

int NodeLM::is_source_node() {
    return _has_source_node;
}

int NodeLM::is_target_node() {
```

```
    return _has_target_node;
}

void NodeLM::update_has_target_node(int value) {
     _has_target_node = value;
}


std::string NodeLM::get_direction(int net_num_to_route, NodeLM* prev_node,
std::string path_num_for_the_net_x, std::string path_num_for_the_net_y ) {
  for (auto const& currentPath: _path_data) {
    if ( (std::get<0>(currentPath) == net_num_to_route) &&
(std::get<2>(currentPath) == prev_node) && ((std::get<1>(currentPath)).first ==
path_num_for_the_net_x) && ((std::get<1>(currentPath)).second ==
path_num_for_the_net_y))
       return std::get<3>(currentPath);
  }
}

int NodeLM::get_num_bends(int net_num_to_route, NodeLM* prev_node, std::string
path_num_for_the_net_x, std::string path_num_for_the_net_y) {
  for (auto const& currentPath: _path_data) {
    if ( (std::get<0>(currentPath) == net_num_to_route) &&
(std::get<2>(currentPath) == prev_node) && ((std::get<1>(currentPath)).first ==
path_num_for_the_net_x) && ((std::get<1>(currentPath)).second ==
path_num_for_the_net_y))
       return std::get<4>(currentPath);
  }
}

std::vector<std::pair<std::pair<std::string,std::string>,NodeLM*>>
NodeLM::get_path(int net_num_to_route) {
  std::vector<std::pair<std::pair<std::string,std::string>,NodeLM*>> vints;
  for (auto const& tuple: _path_data) {
      if ( std::get<0>(tuple) == net_num_to_route )
        vints.push_back(std::make_pair(std::get<1>(tuple),std::get<2>(tuple)));
  }
  return vints;
}

void NodeLM::add_prev_node_data(int net_num_to_route, std::string path_sum_x,
std::string path_sum_y, NodeLM* prev_node, std::string direction, int num_bends,
int is_on_multiple_net_paths, int is_on_already_selected_path) {
     _path_data.push_back(std::make_tuple(net_num_to_route,
std::make_pair(path_sum_x, path_sum_y), prev_node, direction, num_bends,
is_on_multiple_net_paths));
}

void NodeLM::delete_prev_node_data() {
     _path_data.clear();
}

int NodeLM::is_already_on_a_nets_path (int net_num_to_route) {
    for (auto const& currentPath: _path_data) {
      if (std::get<0>(currentPath) != net_num_to_route ) {
         return 1;
      }
    }
    return 0;
}

int NodeLM::is_already_on_a_selected_path () {
    return _is_on_selected_path;
}
```

```
void NodeLM::add_on_selected_path () {
    _is_on_selected_path = 1;
}

void NodeLM::delete_from_selected_path () {
    _is_on_selected_path = 0;
}

std::pair<std::string,NodeLM*> NodeLM::get_path_with_min_bends(int
net_num_to_route, std::string dir) {
     std::pair<std::string,NodeLM*> path_with_min_bends;
     int min_bends = INT_MAX;

     for (auto const& currentPath: _path_data) {
       if ( std::get<0>(currentPath) == net_num_to_route ) {
          if ( std::get<4>(currentPath) < min_bends ) {
             path_with_min_bends = std::make_pair(std::get<3>(currentPath),
std::get<2>(currentPath));
             min_bends = std::get<4>(currentPath);
          } else if ( (std::get<4>(currentPath) == min_bends) &&
(std::get<3>(currentPath) == dir)) {
             path_with_min_bends = std::make_pair(std::get<3>(currentPath),
std::get<2>(currentPath));
             min_bends = std::get<4>(currentPath);

          }
       }
     }
     return path_with_min_bends;
}

std::pair<int,int> NodeLM::get_coordinates(int grid_node_size) {
     int x = (this->x() - grid_node_size - grid_node_size/2) / grid_node_size;
     int y = (this->y() - grid_node_size - grid_node_size/2) / grid_node_size;
     return std::make_pair(x,y);
}

int NodeLM::get_node_num(int grid_node_size, int gridsize) {
    std::pair<int,int> coordinates = get_coordinates(grid_node_size);
    return coordinates.first * gridsize + coordinates.second;
}
```

Lee_Moore_routing.cpp

```
#include <string>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <sstream>
#include <regex>
#include <set>
#include <algorithm>
#include <climits>
#include <cfloat>
#include <typeinfo>
#include <cmath>
#include <SFML/Graphics.hpp>
```

```cpp
#include "Node.h"
#include "NodeLM.h"
#include "read_data_helper.h"
#include "LM_graphHelper.h"
#include "LM_router.h"

int main(int argc, char * argv[]) {

    const int grid_node_size = 40;
    std::string maze_file;
    int debug = 1;
    int gridsize;
    std::vector<std::pair<std::pair<int,int>,std::pair<int,int>>> nets;
    NodeLM* node_data;

    if ( argc == 3 ) {
        maze_file = argv[1];
        std::cout << maze_file << "\n";
        debug = atoi(argv[2]);
        std::cout << debug << "\n";
    } else {
        maze_file = "./maze_tests.txt";
        std::cout << maze_file << "\n";
        debug = 0;
        std::cout << debug << "\n";
    }

    node_data = read_maze_file(maze_file, &gridsize, &nets, grid_node_size,
debug);
    if (debug) {
       std::cout << "\n\nPrinting net sources and sinks from main function:\n";
       for ( std::pair<std::pair<int,int>,std::pair<int,int>> &net_coordinates:
nets ) {
           std::cout << net_coordinates.first.first << ":" <<
net_coordinates.first.second << "::" << net_coordinates.second.first << ":" <<
net_coordinates.second.second << "\n";
       }
       std::cout <<
"*********************************************************\n";
    }

    fillGraphGrid(node_data, gridsize, grid_node_size);
    if ( debug ) {
       for ( int i = 0; i < gridsize; i++ ) {
         for (int j = 0; j < gridsize; j++ ) {
             std::cout << i << ":" << j  << " " << node_data[i * 15 + j].x() <<
":" << node_data[i * 15 + j].y() << "\n\t";
             for (NodeLM* neighbor: node_data[i*15 + j].neighbors()) {
                std::cout << " " << neighbor->x() << ":" << neighbor->y();
             }
             std::cout << "\n";
         }
         std::cout << "\n\n";
       }
    }

    lm_router(&nets, node_data, gridsize * gridsize, gridsize, grid_node_size);
    return 0;
}
```